SIMPLE AND EFFICIENT DISTRIBUTED ALGORITHMS FOR ELECTION IN COMPLETE NETWORKS

Yehuda Afek
Eli Gafni

October 1984
Report No. CSD-840042

SIMPLE AND EFFICIENT DISTRIBUTED ALGORITHMS FOR ELECTION IN COMPLETE NETWORKS

YEHUDA AFEK
ELI GAFNI
Computer Science Department
University of California, Los Angeles, CA 90024

Abstract

The message complexity of an election algorithm in a general network is $O(m+n\cdot\log n)$. Recently, an $O(n\log n)$ election algorithm for a complete network was presented [6]. However, the algorithm is quite complicated in terms of understanding and programming. Here, we present simple and short election algorithms which are as efficient. The suggested algorithms are of practical importance for users who are actually interested in programming the election algorithm on a complete network of processors.

1 Introduction

The election algorithm is an identical program residing in every node in the network. Initially, nodes differ only by their identifiers (*ids*). The nodes elect a leader by exchanging messages. When the message exchange terminates, one node is distinguished from all others.

In [5], an $O(m+n\cdot\log n)$ election algorithm for general networks was presented where m is the number of links and, n is the number of nodes in the network. It was later proven in [1,7] that $O(n\cdot\log n)$ is a lower bound. A lower bound of $O(m)$ is obvious since any untraversed link could be the only link connecting two parts of the network, each holding a separate election algorithm.

In, [6] Korach et. al. showed that the $O(m)$ lower bound does not hold for the special case of a complete network, i.e., when every node has a link to every other node in the network. The authors then presented an election algorithm for complete networks. The algorithm follows that presented in [5] and, in terms of understanding and programming, is as complex.

*This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-82-C-0064. The first author is an IBM Graduate Fellow.

In complete networks, an election algorithm may terminate if all the the neighbors of one node have been visited. This is not the case in the general network. Taking advantage of the simplicity of termination detection, we apply principles suggested in [4,5] for general networks to derive two simple and elegant algorithms (A and B) for election in complete networks. The two algorithms present a tradeoff between time and message complexities. Algorithm A has an $O(n)$ time complexity and $2.773 \cdot n \cdot \log n$ message complexity. Algorithm B has a $2 \cdot n \cdot \log n$ message complexity but an $O(n \cdot \log n)$ time complexity. Analyzing the communication and time complexities of the two algorithms, we derive a third algorithm, algorithm C, which achieves the time complexity of algorithm A and the communication complexity of algorithm B. The principles in [4] have already been successfully applied in [2] to produce an efficient election algorithm in unidirectional networks. These results suggest that the principles behind the presented algorithms are fundamental to the area of distributed algorithms.

2 Algorithms for Election in Complete Networks

2.1 Overview

We present three algorithms (A, B, and C) for election in a complete network. The underlying mechanism for all three algorithms is similar. Each algorithm is initiated by any subset of nodes, each of which spawns a process. Each spawned process tries to capture all the other nodes by successfully traversing in both directions all the links incident to its initiator, called the *base*. The *base* node whose process has succeeded in capturing all its neighbors, elects itself as the leader of the network. To guarantee that only one node is elected, all processes but one are *killed*.

Processes in the different versions of the algorithm use a combination of two variables, *id* and *level*, in order to contest each other. The process *id* is the id of its *base* node. The process *level* is an estimate of the amount of work it has already done. Process *level* could, for example, be some function of the number of nodes it has captured, the number of processes it has killed or both.

To keep track of its owner process, every captured node has two link pointers, *father* and *potential-father*. In general, the *father* pointer points to the link through which the node was most recently captured. The *potential-father* pointer points to the link through which a process which tries to capture the node has arrived.

Every captured node has a *level* variable, which is the highest level process it has observed.

A process that arrives at a node with a larger *level* than its own, kills itself. On the other hand, if the node's *level* is smaller or equal to that of the process, the node's *level* is replaced by the process *level*, which is then said to survive. The surviving process may then try to kill the previous owner of the node. The different criterion which the process uses to kill the previous owner give rise to the different algorithms.

The simplicity of the election algorithms stems from the simplicity of termination detection and process synchronization in a complete network. Termination is easily detected by a *base* node when its process has captured all its neighbors. Synchronization between two conflicting processes is easy since each captured node is at most one hop away from its owning *base* node.

The main difference between the three algorithms is the way that processes determine their *level*. In algorithm A, processes use the number of nodes they have captured as their *level*. In algorithm B, processes use the number of processes they have killed. Among the two algorithms, algorithm A achieves a better time complexity while algorithm B achieves a better communication complexity. In algorithm C, processes use a combination of the above two *level* functions to achieve the message complexity of algorithm B and the time complexity of algorithm A.

2.2 Algorithm A

In this algorithm, the *level* of a process is the number of nodes it has already captured. When a process replaces the *level* of an already captured node, it has to kill the previous owner of the node before it may claim the node to itself.

To resolve ties, the *process id* (*base id*) is appended to the *level* of every process and every node, and comparisons are carried out lexicographically. Nodes initialize their *level* to 0. Processes initialize their *level* to 0 appended by their *base-id*.

A process that arrives at an already captured node v whose *level* is smaller than its own, replaces v's *level* with its own and becomes v's *potential-father*. The *potential-father* process is then sent to the *base* node which owns v. If the process survives at v's *base* node, and no other process became v's *potential-father* in the meanwhile, then it becomes v's *father*.

Figure 1 shows the formal description of algorithm A. In the algorithm, E is the set of edges incident to a node. Every *base* node maintains a list of edges, called *untraversed*, which it's process has not yet traversed in any direction. The untraversed list is used by *base* nodes, live and dead, to decide whether a visiting process is one which tries to capture it or is a conflicting *potential-father*. A process which comes over a link from the untraversed set is a capturing process while a process which arrives through an already traversed link is certainly a disputing *potential-father*. In this way the algorithm has only one type of message.

The algorithm is deadlock free since processes never wait for each other, and the (*level*, *id*) pair is lexicographically increasing along any chain of processes which kill each other.

The time complexity of the algorithm is $O(n)$ since processes never wait for each other and a process which has done more work is never killed by a process which has done less work. Thus, each killed process spent less time, in the worst case, than the process killing it.

```
level := owner-id := 0 ;
untraversed := E ; father := nil ;

    /* The following is performed by initiator nodes */

Base ( id ) :
while ( untraversed ≠ Ø ) do;
      e := any( untraversed ) ;
      send(id,level) on e ;
R:    receive(id',level') over e' ;
      if (id' = id) then          /*process returns from successful capturing.*/
            level := level + 1 ;
            untraversed := untraversed - e ;
      else                        /*another process tries to capture the node.*/
           if (level',id' < level,id)      /* comparison done lexicographically*/
           then  Discard the message, goto R ;
           else  goto Killed ;
end while;

announce(ELECTED,terminate the algorithm) , STOP ;     /* untraversed = Ø */

    /* The following is performed by non-initiator nodes */
Captured:
for_ever do;
   receive(id',level') over e' ;
   case level', id' of :
   (1) level',id' < level,owner-id:
         Discard message ;
   (2) level',id' > level,owner-id :
Killed :  potential-father := e' ;
         level := level' ;
         owner-id := id' ;
          /*Now check if id' is a disputing potential-father or a capturing process*/
         if (e' ∉ untraversed) then send(id',level') over e', goto Captured ;
         if father = nil then  father:=potential-father ;
         send(id',level') over the father link ;
   (3) level', id' = level,owner-id:    /* Potential father has killed the father */
         father := potential-father ;
         send(id', level') over the father link ; /* Return the process to it's base.*/
   end case ;
end for_ever ;
```

FIGURE 1: Algorithm A

The analysis of the communication complexity of the algorithm relies on the following observation: The sets of captured nodes, each owned by a different live process, are disjoint. This is because every node has at most one *father*, and when a node changes a *father*, the previous one is dead. Hence, if one process owns the entire network, it is the only live process in the network.

To prove that the communication complexity of the algorithm is $O(n \cdot \log n)$ we use a Lemma which was introduced in [4].

Lemma 1: For any given k, the number of processes that own $\frac{n}{k}$ nodes or more is, at most, k.

The proof of Lemma 1 relies on the disjointness property. Details can be found in [4].

Corollary 2: The largest process to be killed by another process owns at most $\frac{n}{2}$ nodes, the next largest at most $\frac{n}{3}$, etc.

Since to capture a node a process makes at most 4 hops, we arrive at the following:

Lemma 3: The total cost of algorithm A is $4 \cdot n \cdot \ln\ n$.

proof: The cost incurred by a process of size k is $4 \cdot k$ messages. From corollary 2, the total cost is bounded by $4 \cdot n \cdot \sum_{i=1}^{n} \frac{1}{i}$ messages. Hence, the total message complexity of the algorithm is bounded by $4 \cdot n \cdot \ln n$ $(2.773 \cdot n \cdot \log n)$ messages. ■

Each message of algorithm A contains at most $2 \cdot \log n$ bits.

The principle behind algorithm A is that the node sets owned by different live processes are disjoint. Thus, a process has to verify that the father of every node it tries to capture is dead before claiming the node to itself. This mechanism causes a process, which captures many nodes from another process, to "kill" that other process as many times as the number of nodes it captures from it and hence, causing the factor 4 in the message complexity. In the next algorithm we relax the disjointness requirement, thus enabling a process to claim a node without killing its previous owner. This modification reduces the message complexity to $2 \cdot n \log n$ messages $(2.885 \cdot n \cdot \ln n)$.

2.3 Algorithm B

As in the previous algorithm, the *level* of a process is initialized to 0; however, here we do not append the process id to the process *level*. Whenever a process sees another one with a higher level, it kills itself. If a process sees another one at the same *level* but with a smaller id, it kills the other process and increases its *level* by one. Thus, the *level* of a process is the number of processes it has killed. As in algorithm A, the node's *level* is the highest *level*-process it has seen.

Unlike algorithm A, processes in this algorithm may claim a node of another process without killing the other process. However, a process may replace the *level* of a node only if the node *level* was less than the process *level*, in which case the father pointer of the node is set to the link through which the process arrived.

If the process *level* equals node v's *level* and its id is larger than that of v's owning process, then the process becomes v's *potential-father*. The *potential-father* is then sent to v's father, f, in an attempt to kill it. If another process in the same *level* with an even higher id arrives at v before the *potential-father* returns from f, then this other process is killed. If the *potential-father* survives at f it returns to its *base* (via v which it now owns) and increments its *level* by one. However, if the *potential-father* process arrives at node f and f was killed already, then the *potential-father* kills itself.

To analyze the performance of the algorithm we note that the maximum *level* achievable during its execution is $\log n$. The reason being, that at most half of the processes at level k can go up to level $k+1$. Clearly, every time a node is recaptured its level is increased by at least one. Hence, the total number of captures possible, is at most $n \cdot \log n$. Each capture costs 2 messages which sums to a total of $2 \cdot n \cdot \log n$. The extra message cost incurred by processes which go over *father* links to other *bases* is at most $2 \cdot n$, since each such traversal results in the elimination of one live process. Thus, the total message complexity of the algorithm is bounded by $2 \cdot n \cdot \log n + 2 \cdot n$ and the message length is $\log n + \log\log n$.

The time complexity of the algorithm is $O(n \cdot \log n)$ as can be seen from the following scenario. We will show how it is possible for $\frac{n}{2}$ of the nodes to be captured $\log\frac{n}{2}$ times. The algorithm is started by node, v_0 which captures $\frac{n}{2}$ nodes and is still in level 0. Then, a new node, v_1, spontaneously starts the algorithm, kills v_0, increases its level to 1 and captures the $\frac{n}{2}$ nodes. After v_1 has captured the $\frac{n}{2}$ nodes, two new nodes spontaneously start the algorithm, try to kill each other and the one which survives, v_2, reaches level 1. Node v_2 then kills v_1 and recaptures the $\frac{n}{2}$ nodes at level 2. The scenario continues until node $v_{\log\frac{n}{2}}$ recaptures the entire network and is elected as a leader.

The reason for the high time complexity of algorithm B is that killings are independent of the number of nodes captured by each of the disputing processes. Hence, a process which spent a lot of work (and time) accumulating nodes might be killed by a process which did not spend nearly as much. Although algorithm A does not suffer from this problem, it had the problem that processes could be "killed" many times. In the next algorithm we eliminate both problems by employing both techniques simultaneously in one algorithm.

2.4 Algorithm C

Here we make two modifications to algorithm B in order to achieve a linear time complexity with no increase in the communication cost. First, we combine an estimate of the amount of work spent by each process into the level function of algorithm B. Second, we enable processes with high levels ($>\log n$) to capture many nodes in parallel (in one time unit). We start describing the algorithm with the first modification. The second modification will be introduced during the performance analysis.

```
/* The variable size is for algorithm C only */

      level := size := 0 ;  owner-id := potential-id := 0 ;
      untraversed := E ; father := potential-father := nil ;

/* The following is performed by initiator nodes */

Base ( id ) :
 while ( untraversed ≠ ∅ ) do;
      e := any( untraversed ) ;
      send(level,id) on e ;
R:    receive(level',id') over e' ;
      if (id' = id) then          /*process returns from successful capturing.*/
            level := level'
            untraversed := untraversed - e ;
      else-if (level',id' < level,id) then /* comparison done lexicographically*/
            Discard the message, goto R ;
      else-if e' ∈ untraversed then goto Killed ; /* capturing process from base id'*/
      else send(level',id') over e'; goto Captured ;/*disputing poten.-father process*/
 end while;

announce(ELECTED, terminate the algorithm ), STOP ;    /* untraversed = ∅ */

/* The following is performed by non-initiator nodes */

Captured:
 while (not terminated) do;
     receive(level',id') over e' ;
     if (e' ∉ untraversed) then  Discard message ; /* cannot kill a dead process */
     else
Killed :  case level' of :
       (1) level' < level :  Discard message ;
       (2) level' > level :     /* Replace the father */
              father := e' ; level := level' ; owner-id := id' ;
              potential-id := 0; potential-father := nil ;
              send(level',id') over the father link ;
       (3) level' = level :
              if      (owner-id = 0) then /* id' becomes the father */
                    father := e' ; level := level' ; owner-id := id' ;
                    potential-id := 0; potential-father := nil ;
                    send(level',id') over the father link ;
              else-if (id' < owner-id) then Discard message ;
              else-if ( id' = potential-id ) then
                    father := potential-father ; level' := level' +1 ;
                    owner-id := id' ; potential-id := 0 ; potential-father := nil ;
                    send(level',id') over the father link ;
              else-if there is already a potential-father then Discard message ;
              else  /*  there is no potential-father */
                    potential-id := id' ; potential-father := e' ;
                    send(level',id') over the father link ;
       end case ;
end while ;
```

FIGURE 2: Algorithm B

As before, the *level* of any process or node is initialized to 0. Processes increase their *level* according to two rules. First, as in algorithm B, whenever a process kills another one, it increases its *level* by 1. Second, when the process returns from a successful capturing, the *base* node replaces its *level* with the maximum of *level* and the log of the number of nodes it has already captured.

The algorithm is similar to algorithm B. In this algorithm *base* nodes use a *size* variable to count the number of nodes they have captured. *Base* nodes increase their *size* by one for each node captured by their processes and update their process *level* by *Max*(*level*,log *size*).

The formal description of the algorithm is similar to that of algorithm B. The only change is, to replace the first then clause, within the while loop of a *base* node program to:

```
then
    size:= size+1 ;
    level:= max( level',log size) ;
    untraversed := untraversed - e ;
```

To analyze its performances we will first show that:

Lemma 4: The maximum *level* reachable during any execution of algorithm C is $\log n + \log\log n + 1$.

proof: Let N_i be the total number of processes that reach *level i* during the execution of the algorithm. Consider the maximum number of processes which could possibly pass from *level i*–1 to *level i*. There are two ways in which a process can grow from *level i*–1 to *level i*. First, by capturing 2^{i-1} nodes at *level i*–1, for $i \leq \log n$ and second, by killing another process which is at *level i*–1. We note that N_i is maximized if as many processes as possible pass from *level i*–1 to *level i* by capturing other nodes (i.e., $\frac{n}{2^{i-1}}$) and the rest of the processes (i.e, $N_{i-1} - \frac{n}{2^{i-1}}$) kill each other in pairs. Hence,

$$N_i \leq \frac{(N_{i-1} - \frac{n}{2^{i-1}})}{2} + \frac{n}{2^{i-1}} \qquad (1)$$

Solving (1) for N_i we get:

$$N_i \leq \frac{n}{2^i} \cdot (i+1) \qquad (2)$$

Substituting $N_i = 1$ in (2) and solving for i gives us the maximum *level* which is, $\log n + \log\log n + 1$. ■

Using exactly the same argument as we used for the message complexity of algorithm B we get that the total number of messages of algorithm C is bounded by $2 \cdot n \cdot (\log n + \log\log n + 2)$ where each message contains at most $\log n + \log(\log n + \log\log n)$ bits.

With the above modification it can be shown that the time complexity of algorithm B was reduced to $O(n \cdot \log\log n)$ without any degradation in the big O message complexity. In order to further reduce the time complexity to $O(n)$, processes at level higher than $\log n$ will try to capture $\frac{n}{\log n}$ nodes in parallel. Thus a process which has reached *level* $\log n$ will send messages over $\frac{n}{\log n}$ untraversed links incident to its *base*. Each of these messages carry the (*level*, *id*) of the *base* node. When arriving at the adjacent nodes they compare their *level* to that of the nodes. If the message *level* is higher it replaces the (*level*, *id*) of the node thus becoming the father of the node and returning to the *base* notifying it of a successful capture. If the message *level* is smaller, it returns no message. Finally, if the message *level* is the same as that of the node but the message id is higher, a notification to that effect is sent back to the *base*.

The *base* node waits for all the $\frac{n}{\log n}$ messages. If all the messages indicate a successful capture, the *base* proceeds to the next $\frac{n}{\log n}$ untraversed incident links. If on the other hand, some of them indicate that they have encountered the same *level*, one of them is arbitrarily chosen and a process that behaves as in algorithm B is sent along the link. If the process return, the *base* node increases its *level* and proceeds to the next $\frac{n}{\log n}$ untraversed links (links on which no successful capture was reported are not considered traversed).

To analyze the algorithm with this modification we make two observations: First, the maximum attainable level in the algorithm is still bounded by $\log n + \log\log n + 1$. Second, by substituting $i = \log n$ in equation (2), we find out that the maximum number of processes which reach level $\log n$ is $\log n$.

The last modification has increased the communication complexity of the algorithm by at most $O(n)$ messages. Each node is still captured at most $\log n + \log\log n + 1$ times, however the death of a processes at level greater than $\log n$ might be associated with at most $2 \cdot \frac{n}{\log n}$ messages. Since there are at most $\log n$ such processes the increase due to killings is bounded by $O(n)$.

To show that the time complexity of the algorithm is $O(n)$ we arrange the processes in a rooted tree. Each level of the tree corresponds to the processes that have reached that *level* in the algorithm, i.e., the nodes at *level* i in the tree correspond to the processes that have reached *level* i in the algorithm. The parent of a process at *level* i in the tree is either the process that caused the death of the given process or, the same process at the next *level*. The time delay of the algorithm is the sum of the delays incurred by processes along the path from the first process spontaneously waking up, at *level* 0, to the root.

To evaluate this time delay we note that no process that either survives or is killed at the i^{th} *level*, spends more than 2^{i-1} time units in the i^{th} *level* for $i \leq \log n$. In levels higher than $\log n$ no process spends more than $\log n$ time units since it captures nodes at a rate of $\frac{n}{\log n}$ per time unit. Hence, the total time delay of the algorithm is bounded by $\sum_{i=1}^{\log n} 2^i + \sum_{i=\log n}^{\log n + \log\log n} \log n = n + \log n \cdot \log\log n$. Note that we scale a time unit to be the maximum delay it takes to capture one node, which is a constant.

In the above calculation of the algorithm's time delay we did not include the actual time it takes processes to kill each other. Since, there are, at most, n processes and, no process tries to kill a dead one (unlike algorithm A), this delay is also bounded by $O(n)$.

3 Conclusions

We have presented a sequence of algorithms for an election in a complete graph where each algorithm was devised to circumvent the problems of its predecessors. This sequence concludes with algorithm C, which has time complexity $O(n)$ and message complexity $2 \cdot n \cdot \log n + O(n)$. Yet, the major contribution of the paper is the study of various methods we believe to be recurrent in a number of election algorithms. We have combined these methods to derive an algorithm better than those employing any one of the methods alone. Recently [3] ,we have applied this to derive an $O(\log n)$ time and $O(n \cdot \log n)$ messages, synchronous election algorithm for a complete network. We are now in the process of investigating the ramification of this idea to election in general networks.

References

[1] Greg N. Frederickson and Nancy A. Lynch, "The Impact of Synchronous Communication on the Problem of Electing a Leader in a Ring," (Nov. 83). Detailed Abstract.

[2] Eli Gafni and Yehuda Afek, "Election and Traversal in Unidirectional Networks," in *Proceedings of the ACM Symp. on Principles of Distributed Computing*, Vancouver BC (August 1984).

[3] Eli Gafni, Yehuda Afek, and Leonard Kleinrock, "Fast and message optimal synchronous election algorithms for a complete network," CSD-840041, UCLA, Los Angeles, CA 90024 (October 1984).

[4] Robert G. Gallager, "Finding a Leader in a Network with O(E) + O(N log N) Messages," , M.I.T. (1977). Unpublished Note.

[5] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira, "A Distributed Algorithm for Minimum Weight Spanning Trees," *ACM Trans. Program. Lang. Syst.* **5**, pp.66-77 (Jan 1983).

[6] E. Korach, S. Moran, and S. Zaks, "Tight Lower and Upper Bounds for Some Distributed Algorithms for a Complete Network of Processors," in *Proceedings of the ACM Symp. on Principles of Distributed Computing*, Vancouver BC (August 1984).

[7] J. Pachl, E. Korach, and D. Rotem, "A Technique for Proving Lower Bounds for Distributed Maximum-Finding algorithms," pp. 378-382 in *Proceedings of the 14th Ann. ACM Symp. on Theory of Computing*, San Francisco CA (1982).